Towards a new generation of cryptographic software

Peter Schwabe

Max Planck Institute for Security and Privacy
Bochum, Germany

10. December 2025

A new generation of crypto software?

A new generation of crypto software?

- Post-quantum

A new generation of crypto software?

- Post-quantum
- NIST-PQC was revealing
  - Wrong proofs
  - Hidden assumptions in proofs
  - Proofs that did not apply to the scheme
  - Software that did not implement the scheme
  - Bugs in software
  - (Compiler-introduced) timing leaks

A new generation of crypto software?

- Post-quantum
- NIST-PQC was revealing
  - Wrong proofs
  - Hidden assumptions in proofs
  - Proofs that did not apply to the scheme
  - Software that did not implement the scheme
  - Bugs in software
  - (Compiler-introduced) timing leaks
- PQC migration as a chance
  - Applications *need* new implementations
  - Piggyback on migration to deploy better software

A new generation of crypto software?

- Post-quantum
- NIST-PQC was revealing
  - Wrong proofs
  - Hidden assumptions in proofs
  - Proofs that did not apply to the scheme
  - Software that did not implement the scheme
  - Bugs in software
  - (Compiler-introduced) timing leaks
- PQC migration as a chance
  - Applications *need* new implementations
  - Piggyback on migration to deploy better software

Goal: Strong link between mathematical object and implementation

About 10 years ago. . .

I hope you're well!

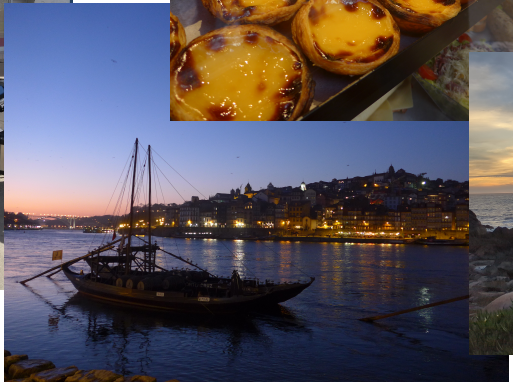
"Gilles Barthe, Ben Laurie, Dan Boneh, Meredith Whittaker, and myself would like to invite you to a workshop on "High Assurance Crypto Software".

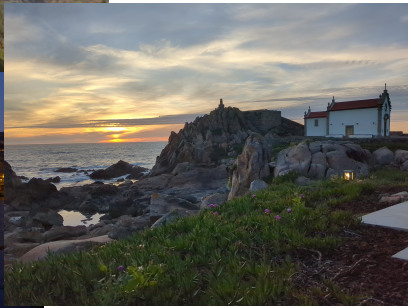
[. . .]

The goal: Bring together the best crypto-implementers and formal-methods verification people from open source, industry, and academia. Introduce them and their projects to each other, and scope and develop practical collaborations that improve verification of crypto code."

—Trevor Perrin, late 2015

Fast forward 4 years (February 2020)





Fast forward 4 years (February 2020)

Goal: Formally verified implementation of Kyber

Almost 6 years later

≈9 papers, 30+ collaborators

Basavesh Ammanaghatta Shivakumar, **Santiago Arranz Olmos**, José Bacelar Almeida, Gustavo Xavier Delerue Marinho Alves, Manuel Barbosa, **Francisca Barros**, Gilles Barthe, Lionel Blatter, Chitchanok Chuengsatiansup, Ignacio Cuevas, François Dupressoir, Luís Esquível, Ruben Gonzalez, **Benjamin Grégoire**, Andreas Hülsing, Vincent Hwang, Jan Jancar, Matthias Kannwischer, **Vincent Laporte**, **Jean-Christophe Léchenet**, Ting-han Lim, Cameron Low, Tiago Oliveira, Hugo Pacheco, Swarn Priya, Miguel Quaresma, Rolfe Schmidt, Antoine Séré, Lucas Tabary-Maujean, Pierre-Yves Strub, Yuval Yarom, Zhiyuan Zhang, **Jieyu Zheng**

Almost 6 years later

≈9 papers, 30+ collaborators

Basavesh Ammanaghatta Shivakumar, **Santiago Arranz Olmos**, José Bacelar Almeida, Gustavo Xavier Delerue Marinho Alves, Manuel Barbosa, **Francisca Barros**, Gilles Barthe, Lionel Blatter, Chitchanok Chuengsatiansup, Ignacio Cuevas, François Dupressoir, Luís Esquível, Ruben Gonzalez, **Benjamin Grégoire**, Andreas Hülsing, Vincent Hwang, Jan Jancar, Matthias Kannwischer, **Vincent Laporte**, **Jean-Christophe Léchenet**, Ting-han Lim, Cameron Low, Tiago Oliveira, Hugo Pacheco, Swarn Priya, Miguel Quaresma, Rolfe Schmidt, Antoine Séré, Lucas Tabary-Maujean, Pierre-Yves Strub, Yuval Yarom, Zhiyuan Zhang, **Jieyu Zheng**

FORMOSA CRYPTO

BOSTON UNIVERSITY

INESCTEC

PQSHIELD

Radboud University

University of BRISTOL

Inria

CASA
CYBER SECURITY IN THE AGE OF LARGE-SCALE ADVERSARIES

ROSENPASS

Universidade do Minho

CRYPTOEXPERTS

SANDBOXAQ

h_da darmstadt university of applied sciences

MAX PLANCK INSTITUTE FOR SECURITY AND PRIVACY

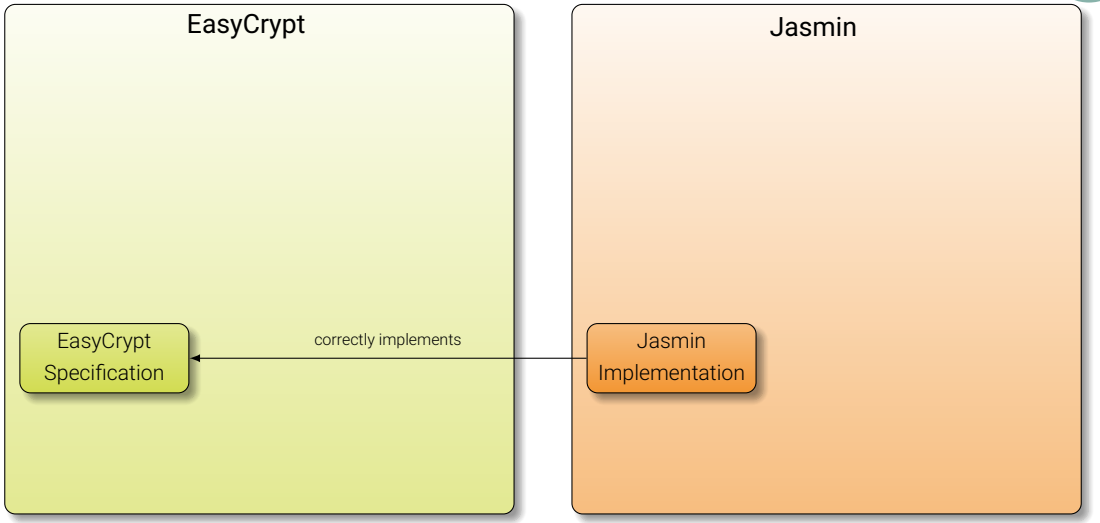
TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

PORTO
FACULDADE DE CIÊNCIAS UNIVERSIDADE DO PORTO

UCLouvain

imdea software

My naive view back in 2020



Jasmin

- Low-level programming language (“assembly in your head”)
- Predictable compilation
  - One line in Jasmin $\rightarrow$ one assembly instruction
  - Jasmin programs are architecture specific
  - Almost full control over generated assembly

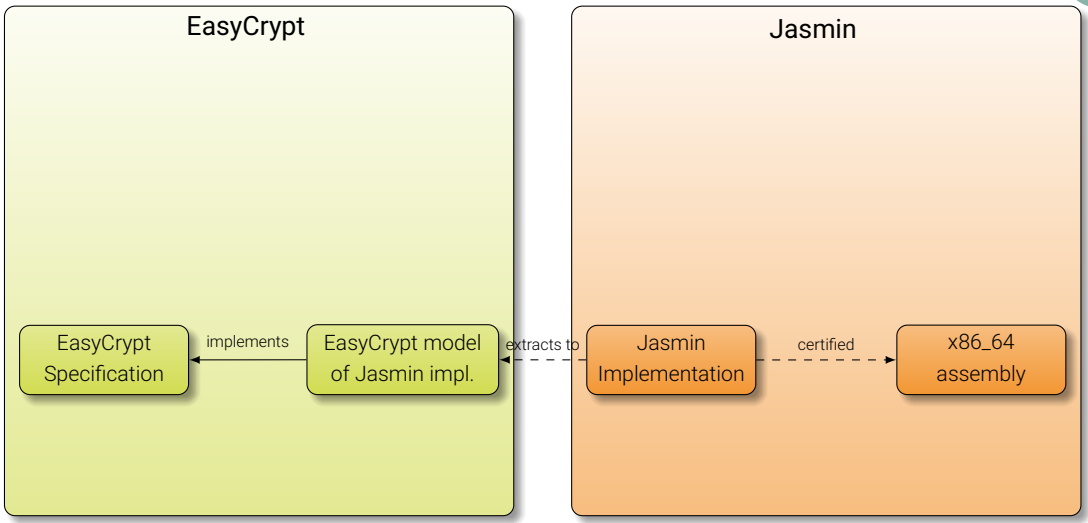
Jasmin

- Low-level programming language (“assembly in your head”)
- Predictable compilation
  - One line in Jasmin $\rightarrow$ one assembly instruction
  - Jasmin programs are architecture specific
  - Almost full control over generated assembly
- High-level features: variables, functions, loops, conditionals. . .
- Register allocation (but no automatic spill-code)

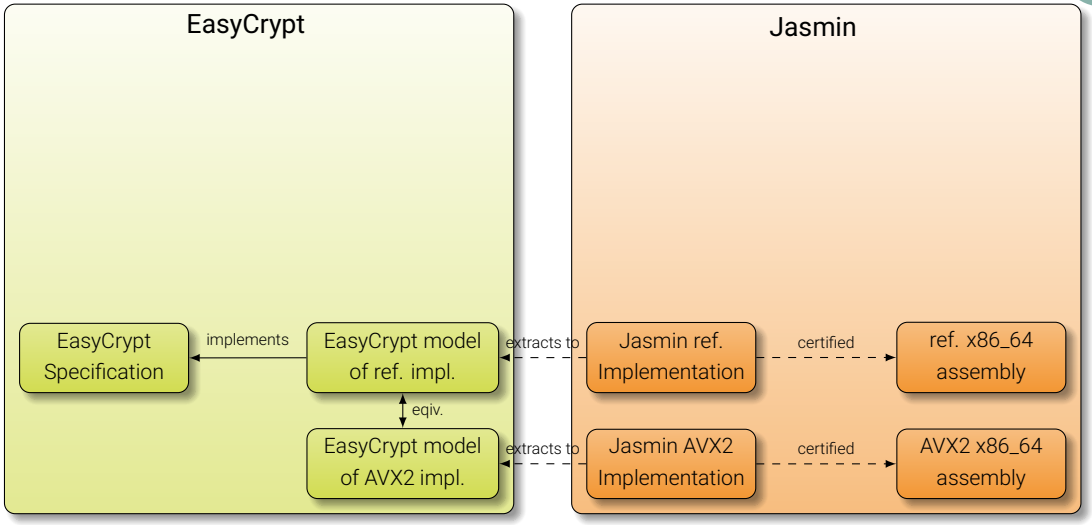
Jasmin

- Low-level programming language (“assembly in your head”)
- Predictable compilation
  - One line in Jasmin $\rightarrow$ one assembly instruction
  - Jasmin programs are architecture specific
  - Almost full control over generated assembly
- High-level features: variables, functions, loops, conditionals. . .
- Register allocation (but no automatic spill-code)
- Important for end-to-end guarantees:
  - *Certified* compilation to assembly (proof in Rocq)
  - Extraction to EasyCrypt
  - Compilation preserves “constant time” (more later)

Refining our high-level view



Refining our high-level view



EasyCrypt

- Interactive verification framework
- Two languages, one functional, one imperative
- Main purpose: Security reductions, game-hopping proofs
  - Security goals and hardness assumptions as probabilistic programs
  - Support for standard arguments (e.g., hybrid arguments)
  - “Formally verify typical pen-and-paper security proofs”

EasyCrypt

- Interactive verification framework
- Two languages, one functional, one imperative
- Main purpose: Security reductions, game-hopping proofs
  - Security goals and hardness assumptions as probabilistic programs
  - Support for standard arguments (e.g., hybrid arguments)
  - “Formally verify typical pen-and-paper security proofs”
- Additional feature: correctness proofs
  - Relate functional and imperative programs
  - Prove equivalence of imperative programs

EasyCrypt

- Interactive verification framework
- Two languages, one functional, one imperative
- Main purpose: Security reductions, game-hopping proofs
  - Security goals and hardness assumptions as probabilistic programs
  - Support for standard arguments (e.g., hybrid arguments)
  - “Formally verify typical pen-and-paper security proofs”
- Additional feature: correctness proofs
  - Relate functional and imperative programs
  - Prove equivalence of imperative programs
- Proofs are manual, but with some automation (e.g., SMT-solver backend)

The specification

- Manuel translation from PDF (Kyber spec, FIPS 203) to EasyCrypt
- Uses both imperative and functional features
- Some interesting tradeoffs, e.g., input/output types for NTT
  - Strong typing: two different isomorphic rings
  - Weaker typing: both just coefficient arrays
  - Weaker typing allows in-place NTT

The specification

- Manuel translation from PDF (Kyber spec, FIPS 203) to EasyCrypt
- Uses both imperative and functional features
- Some interesting tradeoffs, e.g., input/output types for NTT
  - Strong typing: two different isomorphic rings
  - Weaker typing: both just coefficient arrays
  - Weaker typing allows in-place NTT
- Translation and checking could be avoided by **machine-readable standards**

The specification

Algorithm 14 K-PKE.Encrypt($\mathsf{ek}_{\mathsf{PKE}}, m, r$)

Uses the encryption key to encrypt a plaintext message using the randomness r.

Input: encryption key $\mathsf{ek}_{\mathsf{PKE}} \in \mathbb{B}^{384k+32}$.
Input: message $m \in \mathbb{B}^{32}$.
Input: randomness $r \in \mathbb{B}^{32}$.
Output: ciphertext $c \in \mathbb{B}^{32(d_u k + d_v)}$.

$N \leftarrow 0$
$\hat{\mathbf{t}} \leftarrow \mathsf{ByteDecode}_{12}(\mathsf{ek}_{\mathsf{PKE}}[0 : 384k])$
$\rho \leftarrow \mathsf{ek}_{\mathsf{PKE}}[384k : 384k + 32]$
for ($i \leftarrow 0; i < k; i$++)
  for ($j \leftarrow 0; j < k; j$++)
    $\hat{\mathbf{A}}[i, j] \leftarrow \mathsf{SampleNTT}(\rho \| j \| i)$
  end for
end for
for ($i \leftarrow 0; i < k; i$++)
  $\mathbf{y}[i] \leftarrow \mathsf{SamplePolyCBD}_{\eta_1}(\mathsf{PRF}_{\eta_1}(r, N))$
  $N \leftarrow N + 1$
end for
for ($i \leftarrow 0; i < k; i$++)
  $\mathbf{e}_1[i] \leftarrow \mathsf{SamplePolyCBD}_{\eta_2}(\mathsf{PRF}_{\eta_2}(r, N))$
  $N \leftarrow N + 1$
end for
$e_2 \leftarrow \mathsf{SamplePolyCBD}_{\eta_2}(\mathsf{PRF}_{\eta_2}(r, N))$
$\hat{\mathbf{y}} \leftarrow \mathsf{NTT}(\mathbf{y})$
$\mathbf{u} \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{A}}^{\top} \circ \hat{\mathbf{y}}) + \mathbf{e}_1$
$\mu \leftarrow \mathsf{Decompress}_1(\mathsf{ByteDecode}_1(m))$
$v \leftarrow \mathsf{NTT}^{-1}(\hat{\mathbf{t}}^{\top} \circ \hat{\mathbf{y}}) + e_2 + \mu$
$c_1 \leftarrow \mathsf{ByteEncode}_{d_u}(\mathsf{Compress}_{d_u}(\mathbf{u}))$
$c_2 \leftarrow \mathsf{ByteEncode}_{d_v}(\mathsf{Compress}_{d_v}(v))$
return $c \leftarrow (c_1 \| c_2)$

```
proc enc_derand(pk : pkey, m : plaintext, r : W8.t Array32.t) : ciphertext = {
  (tv,rho) ← pk;
  _N ← 0;
  thati ← EncDec.decode12_vec(tv);
  that ← ofipolyvec thati;
  i ← 0;
  while (i < kvec) {
    j ← 0;
    while (j < kvec) {
      XOF(O).init(rho,W8.of_int i, W8.of_int j);
      c ← Parse(XOF,O).sample();
      aT[(i,j)] ← c;
      j ← j + 1;
    }
    i ← i + 1;
  }
  i ← 0;
  while (i < kvec) {
    c ← CBD2(PRF).sample(r,_N);
    yv ← set yv i c;
    _N ← _N + 1;
    i ← i + 1;
  }
  i ← 0;
  while (i < kvec) {
    c ← CBD2(PRF).sample(r,_N);
    e1 ← set e1 i c;
    _N ← _N + 1;
    i ← i + 1;
  }
  e2 ← CBD2(PRF).sample(r,_N);
  yhat ← nttv yv;
  u ← invnttv (ntt_mmul aT yhat) + e1;
  mp ← EncDec.decode1(m);
  v ← invntt (ntt_dotp that yhat) + e2 + decompress_poly 1 mp;
  c1 ← EncDec.encode10_vec(compress_polyvec 10 u);
  c2 ← EncDec.encode4(compress_poly 4 v);
  return (c1,c2);
}
```

Mechanizing the security proof

Goal: Link specification to IND-CCA notion

Mechanizing the security proof

Goal: Link specification to IND-CCA notion

Step 1: IND-CPA security of K-PKE

- Typical game-hopping proof for abstract LWE-based PKE
- Refine to concrete parameters of K-PKE
- Reduction from *hashed MLWE*

Mechanizing the security proof

Goal: Link specification to IND-CCA notion

Step 1: IND-CPA security of K-PKE

- Typical game-hopping proof for abstract LWE-based PKE
- Refine to concrete parameters of K-PKE
- Reduction from *hashed MLWE*

Step 2: Decryption-failure bounds for K-PKE

- Original analysis assumes compression of uniform coefficients
- Cannot rely on MLWE assumption in correctness game
- Move from adversarial notion to statistic notion
- Distribution too complex to fully compute

Mechanizing the security proof

Goal: Link specification to IND-CCA notion

Step 1: IND-CPA security of K-PKE

- Typical game-hopping proof for abstract LWE-based PKE
- Refine to concrete parameters of K-PKE
- Reduction from *hashed MLWE*

Step 2: Decryption-failure bounds for K-PKE

- Original analysis assumes compression of uniform coefficients
- Cannot rely on MLWE assumption in correctness game
- Move from adversarial notion to statistic notion
- Distribution too complex to fully compute

Step 3: IND-CCA security of ML-KEM

- Prove IND-CCA for MLKEM_OP
- Central difference: SHA3-512 replaced by RO

Linking the spec to the code

Spec $\leftrightarrow$ ref impl.

- Montgomery and Barrett reductions
- Lazy reductions (bounds checking!)
- "Funny" code, e.g., for `a = a/3329`

```
a *= 80635;
a >>= 28;
```

- Magic values in NTT code
- Link functional and imperative code

Linking the spec to the code

Spec ↔ ref impl.

- Montgomery and Barrett reductions
- Lazy reductions (bounds checking!)
- "Funny" code, e.g., for `a = a/3329`

```
a *= 80635;
a >>= 28;
```

- Magic values in NTT code
- Link functional and imperative code

ref. impl. ↔ AVX2 impl.

- Optimizations *across* functions
- Different internal data representations
- Non-word-size operations (compress)
- Complex vectorized rejection sampling

Linking the spec to the code

Spec ↔ ref impl.

- Montgomery and Barrett reductions
- Lazy reductions (bounds checking!)
- "Funny" code, e.g., for `a = a/3329`

```
a *= 80635;
a >>= 28;
```

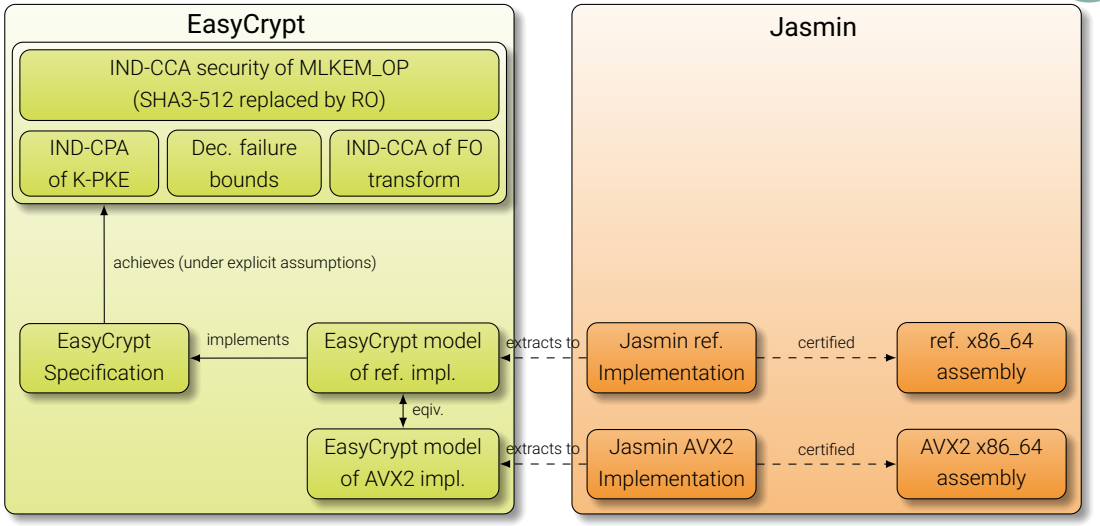
- Magic values in NTT code
- Link functional and imperative code

ref. impl. ↔ AVX2 impl.

- Optimizations *across* functions
- Different internal data representations
- Non-word-size operations (compress)
- Complex vectorized rejection sampling

- Massive manual effort with quite some student frustration
- Significant improvement 2025:
  - *automated* circuit-equivalence checking
  - integrated in EasyCrypt, interleave with deductive reasoning
  - faster verification, more robust proofs, easier proof maintenance

So, where are we?



Safety

The old way

- Static safety check:
  `jasminc -checksafety`
- Great when it works
- Takes a long time (not modular)
- Often fails for safe code

Safety

The old way

- Static safety check:
  `jasminc -checksafety`
- Great when it works
- Takes a long time (not modular)
- Often fails for safe code

The new way

- Master's thesis by **Francisca Barros**
- Modular design
- Safety contracts and assertions in Jasmin
- Automatic discharge of assertions
- Prove remaining assertions in EasyCrypt

```
fn _gen_matrix_avx2
( reg mut ptr u16[MLKEM_K * MLKEM_K * MLKEM_N] matrix
, reg const ptr u8[32] rho
, #spill_to_mmx reg u64 transposed
) -> reg ptr u16[MLKEM_K * MLKEM_K * MLKEM_N]
requires {is_arr_init(rho,0,32) && 0<= transposed && transposed <= 1}
ensures  {is_arr_init(result.0,0,MLKEM_K * MLKEM_K * MLKEM_N * 2)}
{
  ...
}
```

Security – “constant time”

- Enforce constant-time on Jasmin source level
- Every piece of data is either `secret` or `public`
- Flow of secret information is traced by type system

  “Any operation with a secret input produces a secret output”

Security – “constant time”

- Enforce constant-time on Jasmin source level
- Every piece of data is either `secret` or `public`
- Flow of secret information is traced by type system

  “Any operation with a secret input produces a secret output”
- Branch conditions and memory indices need to be `public`

Security – “constant time”

- Enforce constant-time on Jasmin source level
- Every piece of data is either `secret` or `public`
- Flow of secret information is traced by type system

  “Any operation with a secret input produces a secret output”
- Branch conditions and memory indices need to be `public`
- In principle can do this also in, e.g., Rust (`secret_integers` crate)
- Jasmin compiler preserves constant-time!

Security – “constant time”

- Enforce constant-time on Jasmin source level
- Every piece of data is either `secret` or `public`
- Flow of secret information is traced by type system

  “Any operation with a secret input produces a secret output”
- Branch conditions and memory indices need to be `public`
- In principle can do this also in, e.g., Rust (`secret_integers` crate)
- **Jasmin compiler preserves constant-time!**
- Explicit `#declassify` primitive to move from `secret` to `public`
- `#declassify` creates a proof obligation!

Future-proof “constant time”

- Some arithmetic instructions also leak through timing
- Example: `DIV` (exploited in “KyberSlash”, CHES 2025)
- Which instructions are safe to use, *also for future CPUs*?

Future-proof “constant time”

- Some arithmetic instructions also leak through timing
- Example: `DIV` (exploited in “KyberSlash”, CHES 2025)
- Which instructions are safe to use, *also for future CPUs*?
- Intel Core Security Team, 2022: Data Operand Independent Timing (DOIT)
  - Subset of x86_64 instructions **guaranteed** to not leak through timing
  - Requires switching CPU to DOIT mode

Future-proof “constant time”

- Some arithmetic instructions also leak through timing
- Example: `DIV` (exploited in “KyberSlash”, CHES 2025)
- Which instructions are safe to use, *also for future CPUs*?
- Intel Core Security Team, 2022: Data Operand Independent Timing (DOIT)
  - Subset of x86_64 instructions **guaranteed** to not leak through timing
  - Requires switching CPU to DOIT mode
- Support for DOIT in Jasmin compiler
  - Use only DOIT instructions on secret inputs
  - No serious limitation for most *optimized* code
  - All relevant vector instructions are DOIT

Spectre v1 (“Speculative bounds-check bypass”)


```
stack u8[10] public;
stack u8[32] secret;
reg u8 t;
reg u64 r, i;

i = 0;
while(i < 10) {
  t = public[(int) i] ;
  r = leak(t);
  ...
}
```

Countermeasures

Fencing

- Can prevent speculation through **barriers** (LFENCE)
- Protecting *all* branches is possible but costly

Countermeasures

Fencing

- Can prevent speculation through **barriers** (LFENCE)
- Protecting *all* branches is possible but costly

Speculative Load Hardening

- Idea: maintain misprediction predicate `ms` (in a register)
- At every branch use arithmetic to update predicate
- Option 1: Mask every loaded value with `ms`
- Option 2: Mask every address with `ms`
- Effect: during misspeculation “leak” constant value

Countermeasures

Fencing

- Can prevent speculation through **barriers** (LFENCE)
- Protecting *all* branches is possible but costly

Speculative Load Hardening

- Idea: maintain misprediction predicate `ms` (in a register)
- At every branch use arithmetic to update predicate
- Option 1: Mask every loaded value with `ms`
- Option 2: Mask every address with `ms`
- Effect: during misspeculation “leak” constant value
- Implemented in LLVM since version 8
  - Still large performance overhead
  - No formal guarantees of security

Selective SLH

Do we need to mask/protect all loads?

Selective SLH

Do we need to mask/protect all loads?

- No need to mask loads into registers that never enter leaking instructions

Selective SLH

Do we need to mask/protect all loads?

- No need to mask loads into registers that never enter leaking instructions
- `secret` registers never enter leaking instructions!
- Obvious idea: mask only loads into `public` registers

Extending the type system

- Type system gets three security levels:
  - `secret:` secret
  - `public:` public, also during misspeculation
  - `transient:` public, but possibly secret during misspeculation

Extending the type system

- Type system gets three security levels:
  - `secret:` secret
  - `public:` public, also during misspeculation
  - `transient:` public, but possibly secret during misspeculation
- Maintain misspeculation flag `ms`:
  - `ms = #init_msf()`: Translate to `LFENCE`, set `ms` to zero
  - `ms = #set_msf(b, ms)`: Set `ms` according to branch condition `b`
  - Branches invalidate `ms`

Extending the type system

- Type system gets three security levels:
  - `secret:` secret
  - `public:` public, also during misspeculation
  - `transient:` public, but possibly secret during misspeculation
- Maintain misspeculation flag `ms`:
  - `ms = #init_msf()`: Translate to `LFENCE`, set `ms` to zero
  - `ms = #set_msf(b, ms)`: Set `ms` according to branch condition `b`
  - Branches invalidate `ms`
- Two operations to lower level:
  - `x = #protect(x, ms)`: Go from `transient` to `public`
  - `#protect` translates to mask by `ms`

Extending the type system

- Type system gets three security levels:
  - `secret:` secret
  - `public:` public, also during misspeculation
  - `transient:` public, but possibly secret during misspeculation
- Maintain misspeculation flag `ms`:
  - `ms = #init_msf()`: Translate to `LFENCE`, set `ms` to zero
  - `ms = #set_msf(b, ms)`: Set `ms` according to branch condition `b`
  - Branches invalidate `ms`
- Two operations to lower level:
  - `x = #protect(x, ms)`: Go from `transient` to `public`
  - `#protect` translates to mask by `ms`
  - `#declassify r`: Go from `secret` to `transient`
  - `#declassify` requires cryptographic proof/argument

Extending the type system

- Type system gets three security levels:
  - `secret:` secret
  - `public:` public, also during misspeculation
  - `transient:` public, but possibly secret during misspeculation
- Maintain misspeculation flag `ms`:
  - `ms = #init_msf()`: Translate to `LFENCE`, set `ms` to zero
  - `ms = #set_msf(b, ms)`: Set `ms` according to branch condition `b`
  - Branches invalidate `ms`
- Two operations to lower level:
  - `x = #protect(x, ms)`: Go from `transient` to `public`
  - `#protect` translates to mask by `ms`
  - `#declassify r`: Go from `secret` to `transient`
  - `#declassify` requires cryptographic proof/argument
- Still: allow branches and indexing only for `public`
- Also, allow non-DOIT instructions only on `public` inputs

The special case of crypto

- We know what inputs are **secret** and what inputs are **public**
- Most of the state is actually **secret**
- Most loads do not need `protect`!

The special case of crypto

- We know what inputs are **secret** and what inputs are **public**
- Most of the state is actually **secret**
- Most loads do not need `protect`!
- Even better: mark additional inputs as **secret**
- No cost if those inputs don't flow into leaking instructions

The special case of crypto

- We know what inputs are **secret** and what inputs are **public**
- Most of the state is actually **secret**
- Most loads do not need `protect`!
- Even better: mark additional inputs as **secret**
- No cost if those inputs don't flow into leaking instructions
- Even better: Spills don't need `protect` if there is no branch between store and load

The special case of crypto

- We know what inputs are **secret** and what inputs are **public**
- Most of the state is actually **secret**
- Most loads do not need `protect`!
- Even better: mark additional inputs as **secret**
- No cost if those inputs don't flow into leaking instructions
- Even better: Spills don't need `protect` if there is no branch between store and load
- Even better: "Spill" public data to MMX registers instead of stack

If there's a Spectre v1...

- Spectre v2: indirect branches (not supported by Jasmin)
- Spectre v3, aka "Meltdown": fix in HW and firmware
- Spectre v4: speculative store bypass (disable with SSBD)

If there's a Spectre v1...

- Spectre v2: indirect branches (not supported by Jasmin)
- Spectre v3, aka "Meltdown": fix in HW and firmware
- Spectre v4: speculative store bypass (disable with SSBD)
- Spectre-RSB: function return speculates *anywhere*
- Attacker can choose to speculate right behind any defense!

If there's a Spectre v1...

- Spectre v2: indirect branches (not supported by Jasmin)
- Spectre v3, aka "Meltdown": fix in HW and firmware
- Spectre v4: speculative store bypass (disable with SSBD)
- **Spectre-RSB**: function return speculates *anywhere*
- Attacker can choose to speculate right behind any defense!
- Solution in Jasmin:
  - Rewrite all returns through "branch table"
  - Implement branch table through *conditional* branches
  - Speculate only to one of the call sites
  - At call sites all **public** values become **transient**
  - Use `#protect` and `ms`

Clearing sensitive data

"...A cryptographic module shall provide methods to zeroize all plaintext secret and private cryptographic keys"

—FIPS 140-3, Section 9.7.A

Clearing sensitive data

"...A cryptographic module shall provide methods to zeroize all plaintext secret and private cryptographic keys"

—FIPS 140-3, Section 9.7.A

Goal of zeroization

Scrub all (sensitive) data from memory (stack) and registers when crypto routine returns.

Clearing sensitive data

"...A cryptographic module shall provide methods to zeroize all plaintext secret and private cryptographic keys"

—FIPS 140-3, Section 9.7.A

Goal of zeroization

Scrub all (sensitive) data from memory (stack) and registers when crypto routine returns.

Failure modes

0. Don't perform any zeroization

Clearing sensitive data

"...A cryptographic module shall provide methods to zeroize all plaintext secret and private cryptographic keys"

—FIPS 140-3, Section 9.7.A

Goal of zeroization

Scrub all (sensitive) data from memory (stack) and registers when crypto routine returns.

Failure modes

0. Don't perform any zeroization
1. Dead-store elimination

Clearing sensitive data

"...A cryptographic module shall provide methods to zeroize all plaintext secret and private cryptographic keys"

—FIPS 140-3, Section 9.7.A

Goal of zeroization

Scrub all (sensitive) data from memory (stack) and registers when crypto routine returns.

Failure modes

0. Don't perform any zeroization
1. Dead-store elimination
2. Only API-level stack zeroization

Clearing sensitive data

"...A cryptographic module shall provide methods to zeroize all plaintext secret and private cryptographic keys"

—FIPS 140-3, Section 9.7.A

Goal of zeroization

Scrub all (sensitive) data from memory (stack) and registers when crypto routine returns.

Failure modes

0. Don't perform any zeroization
1. Dead-store elimination
2. Only API-level stack zeroization
3. Don't scrub source-level invisible data

Clearing sensitive data

"...A cryptographic module shall provide methods to zeroize all plaintext secret and private cryptographic keys"

—FIPS 140-3, Section 9.7.A

Goal of zeroization

Scrub all (sensitive) data from memory (stack) and registers when crypto routine returns.

Failure modes

0. Don't perform any zeroization
1. Dead-store elimination
2. Only API-level stack zeroization
3. Don't scrub source-level invisible data
4. Mis-estimate stack space when scrubbing from caller

Clearing sensitive data (ctd.)

Solution in Jasmin compiler

Zeroize used stack space and registers when returning from export function

Clearing sensitive data (ctd.)

Solution in Jasmin compiler

Zeroize used stack space and registers when returning from export function

- Make use of multiple features of Jasmin:
  - Compiler has global view
  - All stack usage is known at compile time
  - Entry/return point is clearly defined

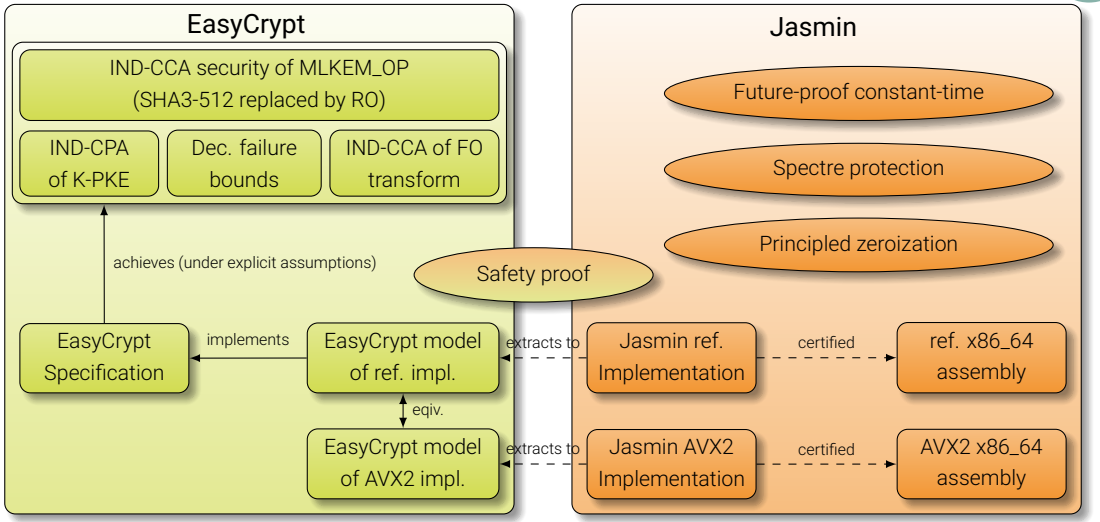
Clearing sensitive data (ctd.)

Solution in Jasmin compiler

Zeroize used stack space and registers when returning from export function

- Make use of multiple features of Jasmin:
  - Compiler has global view
  - All stack usage is known at compile time
  - Entry/return point is clearly defined
- Performance overhead for Kyber768 (on Comet Lake):
  - 0.59% for Keypair
  - 0.24% for Encaps
  - 1.04% for Decaps

The next generation of crypto software



Is it fast?

Cycles for ML-KEM-768

CPU	Implementation	keypair	enc	dec
8700K	Jasmin AVX2*	40134	40599	43437
	pq-crystals	39722	39761	46161
11700K	Jasmin AVX2*	37458	37798	39970
	pq-crystals	36958	38082	42566
13900K	Jasmin AVX2*	34732	35212	43784
	pq-crystals	31448	32090	36064

*with Spectre-v1 (without Spectre-RSB) protections

Stay tuned

Work in progress (selection)

- Deployment in Signal's contact discovery (RWC talk upcoming!)
- Integrate Spectre-RSB protections
- Implement in **crypto agent** process
- Extend to more architectures and more primitives (next up: ML-DSA)
- Interface to super-optimizers (e.g., CryptOpt, SLOTHY)
- Improve usability/scalability of tools

Longer-term plans (selection)

- Guarantee preservation of speculative constant time
- Masked implementations
- Machine-readable standards (?)

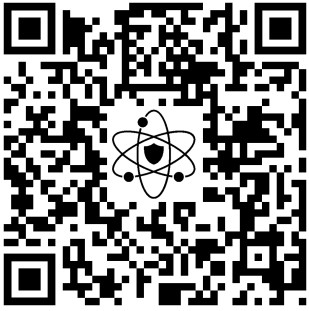
References

- Barbosa, Kannwischer, Lim, Schwabe, Strub. *Formally Verified Correctness Bounds for Lattice-Based Cryptography.* ACM CCS 2025.
- Almeida, Marinho Alves, Barbosa, Barthe, Esquível, Hwang, Oliveira, Pacheco, Schwabe, Strub. *Faster Verification of Faster Implementations: Combining Deductive and Circuit-Based Reasoning in EasyCrypt.* IEEE S&P 2025.
- Arranz Olmos, Barthe, Grégoire, Jancar, Laporte, Oliveira, Schwabe. *Let's DOIT: Using Intel's Extended HW/SW Contract for Secure Compilation of Crypto Code.* TCHES 2025-3.
- Arranz Olmos, Barthe, Chuengsatiansup, Grégoire, Laporte, Oliveira, Schwabe, Yarom, Zhang. *Protecting Cryptographic Code Against Spectre-RSB (and, in Fact, All Known Spectre Variants).* ASPLOS 2025.
- Barbosa, Schwabe. *Kyber terminates.* Polynesian Journal of Mathematics, vol. 1, issue 6 (2024).

References

- Almeida, Arranz Olmos, Barbosa, Barthe, Dupressoir, Grégoire, Laporte, Léchenet, Low, Oliveira, Pacheco, Quaresma, Schwabe, Strub. *Formally verifying Kyber – Episode V: Machine-checked IND-CCA security and correctness of ML-KEM in EasyCrypt.* CRYPTO 2024.
- Arranz Olmos, Barthe, Gonzalez, Grégoire, Laporte, Léchenet, Oliveira, Schwabe. *High-assurance zeroization.* TCHES 2024-1.
- Almeida, Barbosa, Barthe, Grégoire, Laporte, Léchenet, Oliveira, Pacheco, Quaresma, Schwabe, Séré, Strub. *Formally verifying Kyber – Episode IV: Implementation Correctness.* TCHES 2023-3.
- Ammanaghatta Shivakumar, Barthe, Grégoire, Laporte, Oliveira, Priya, Schwabe, Tabary-Maujean. *Typing High-Speed Cryptography against Spectre v1.* IEEE S&P 2023.

Thank you!



https://github.com/pq-code-package/mlkem-libjade

https://formosa-crypto.org